A COMPARISON OF GPU PROGRAMMING MODELS

Trey White, Frontier Center of Excellence
2020 Performance, Portability, and Productivity in HPC Forum
COMMUNICATION FROM NEKBONE BENCHMARK

- (3D) Grid of spectral elements
- That share faces that must be summed
- Partitioned across MPI tasks
- With contiguous buffers for MPI
• (3D) Grid of spectral elements
• That share faces that must be summed
• Partitioned across MPI tasks
• With contiguous buffers for MPI
COMMUNICATION FROM NEKBONE BENCHMARK

- (3D) Grid of spectral elements
- That share faces that must be summed
- Partitioned across MPI tasks
- With contiguous buffers for MPI
COMMUNICATION FROM NEKBONE BENCHMARK

- (3D) Grid of spectral elements
- That share faces that must be summed
- Partitioned across MPI tasks
- With contiguous buffers for MPI
COMMUNICATION FROM NEKBONE BENCHMARK

- Overlap
  - Internal face sums
  - Copies into send buffers
  - MPI communication
  - Outer face sums
• Overlap
  • **Internal face sums**
  • Copies into send buffers
  • **MPI communication**
  • Outer face **sums**
COMMUNICATION FROM NEKBONE BENCHMARK

- Overlap
  - Internal face sums
  - Copies into send buffers
  - MPI communication
  - Outer face *sums*
IMPLEMENTATIONS

- Fortran with OpenMP 4.5
- C++ with OpenMP 4.5
- C++ with Cuda
- C++ with Hip
- C++ with Kokkos
- C++ with Raja and Umpire
FEATURE COMPARISON

- Multi-dimensional arrays
- Elements stored on GPUs
- MPI buffers stored on GPUs
- Translating loops to kernels
- Running kernels in parallel
FEATURE COMPARISON

- Multi-dimensional arrays
  - Elements stored on GPUs
  - MPI buffers stored on GPUs
  - Translating loops to kernels
  - Running kernels in parallel
MULTI-DIMENSIONAL ARRAYS

- Fortran with OpenMP: native
- C++ with OpenMP
  - Current implementations don't map objects effectively
    - Forces C-like code
    - Cannot yet port existing array classes
- Cuda and Hip
  - Do it yourself
  - Good support for C++ modularity features:
    - constructors, destructors, operator overloading, lambdas
  - Port existing array classes
MULTI-DIMENSIONAL ARRAYS

• **Kokkos::View**
  • Defaults to column major for GPUs
  • Owns memory, includes copy methods
  • Intended to replace existing array classes?

• **RAJA::View**
  • Defaults to row major
  • Wraps a provided pointer
  • Can use Umpire to allocate and copy
  • Not a stand-alone array class
  • Useful for porting existing array classes
FEATURE COMPARISON

- Multi-dimensional arrays
- **Elements stored on GPUs**
- MPI buffers stored on GPUs
- Translating loops to kernels
- Running kernels in parallel
ELEMENTS STORED ON GPUs: FORTRAN WITH OPENMP

- Map clauses (no need for array dimensions)
  
  !$omp target data map(u)
  
  ...
  
  !$omp end target data

- Eliminate implicit copies by enclosing larger regions in `target data`
- Nest `target data` regions, may be lower in call stack
- Allows incremental tuning and debugging

**WARNING:** Changes in caller code can change local OpenMP semantics
Current implementations of `map` clauses have limitations
- Support for raw pointers only, not array classes
- Need explicit array bounds
- Need local copies of `this` member variables

```c++
// `du__` and `ue__` are member variables
// Make local copies
int *const du = du__;
double *const ue = ue__;
#pragma omp target data map(to:du[6]) map(ue[:du[5]])
```
ELEMENTS STORED ON GPUs

- **Kokkos::View**
  - On GPU by default
  - Can create explicitly host-allocated **Kokkos::View**
  - Supports explicit copies between GPU and host views

- Cuda, Hip, Raja
  - Allocate GPU memory
  - Do explicit copies to/from host pointers
  - Raja: Can use Umpire to avoid GPU-specific code
FEATURE COMPARISON

- Multi-dimensional arrays
- Elements stored on GPUs
- **MPI buffers stored on GPUs**
- Translating loops to kernels
- Running kernels in parallel
MPI BUFFERS STORED ON GPUs: FORTRAN WITH OPENMP

- MPI buffers can be module variables
- Init procedure allocates them on GPU, deallocating first if necessary

```fortran
!$omp taskwait
if (initted_) then
  !$omp target exit data map(delete:corner_)
  ! deallocate host arrays
  ...
end if
initted_ = .true.
! allocate and initialize host arrays
  ...
  !$omp target enter data map(to:corner_)
  ...
```
MPI BUFFERS STORED ON GPUs: C++ WITH OPENMP

- MPI buffers can be class members
- Constructor maps `to`, destructor maps `delete`

```cpp
static void alloc(double *&p, const long n)
{
    double *const q = new double[n];
    memset(q, 0, n*sizeof(double));
    #pragma omp target enter data map(to:q[:n])
    p = q;
}
```

```cpp
Faces::Faces(...):
    ...
    xfr0_(nullptr), xfr1_(nullptr)
{
    ...
    alloc(xfr0_, dxf_[3]);
    alloc(xfr1_, dxf_[3]);
    ...
```
MPI BUFFERS STORED ON GPUS: FORTRAN AND C++ WITH OPENMP

- MPI access to GPU pointers through `use_device_ptr`

```c
!$omp target data map(u) &
!$omp use_device_ptr(xface_,yface_,zface_,xedge_,yedge_,zedge_,corner_)
...

#pragma omp target data use_device_ptr(xfr0,xfr1,yfr0,yfr1,zfr0,zfr1)
{
    MPI_Irecv(xfr0,dxf_[3],MPI_DOUBLE,iface_[0],tag,MPI_COMM_WORLD,reqr_+0);
    MPI_Irecv(xfr1,dxf_[3],MPI_DOUBLE,iface_[1],tag,MPI_COMM_WORLD,reqr_+1);
    MPI_Irecv(yfr0,dyf_[3],MPI_DOUBLE,iface_[2],tag,MPI_COMM_WORLD,reqr_+2);
    MPI_Irecv(yfr1,dyf_[3],MPI_DOUBLE,iface_[3],tag,MPI_COMM_WORLD,reqr_+3);
    MPI_Irecv(zfr0,dzf_[3],MPI_DOUBLE,iface_[4],tag,MPI_COMM_WORLD,reqr_+4);
    MPI_Irecv(zfr1,dzf_[3],MPI_DOUBLE,iface_[5],tag,MPI_COMM_WORLD,reqr_+5);
}
```
Contiguous MPI buffers are on GPU
Array objects return raw GPU pointer
MPI uses GPU pointers directly

```
MPI_Isend(xes0.data(), nedge_[0], MPI_DOUBLE, iedge_[0], tag, 
          MPI_COMM_WORLD, reqs_+6);
```
FEATURE COMPARISON

- Multi-dimensional arrays
- Elements stored on GPUs
- MPI buffers stored on GPUs
- **Translating loops to kernels**
- Running kernels in parallel
TRANSLATING LOOPS TO KERNELS

• Fortran with OpenMP
  • Easy! For \( N \) nested loops:
    \[
    !$\text{omp target teams distribute parallel do simd collapse}(N)\]
  • Cray Fortran compiler just needs \textit{distribute}, others look for \textit{parallel do}

• C++ with OpenMP
  • Easy! For \( N \) nested loops:
    \[
    \#\text{pragma omp target teams distribute parallel for simd collapse}(N)\]
  • Implementations pretty much agree on emphasizing \textit{parallel for}

• Cuda, Hip
  • Do it yourself
Kokkos::parallel_for("internal corners",
Kokkos::MDRangePolicy<Kokkos::Rank<3>>({1,1,1},{mx,my,mz}),
KOKKOS_LAMBDA(const int jx, const int jy, const int jz) {
    u(0,0,0,jx,jy,jz) += u(nm1,0,0,jx-1,jy,jz)
    + u(0,nm1,0,jx,jy-1,jz)+u(nm1,nm1,0,jx-1,jy-1,jz)
    + u(0,0,nm1,jx,jy,jz-1)+u(nm1,0,nm1,jx-1,jy,jz-1)
    + u(0,nm1,nm1,jx,jy-1,jz-1)+u(nm1,nm1,nm1,jx-1,jy-1,jz-1);
    u(nm1,0,0,jx-1,jy,jz) = u(0,nm1,0,jx,jy-1,jz)
    = u(nm1,nm1,0,jx-1,jy-1,jz) = u(0,0,nm1,jx,jy,jz-1)
    = u(nm1,0,nm1,jx-1,jy,jz-1) = u(0,nm1,nm1,jx,jy-1,jz-1)
    = u(nm1,nm1,nm1,jx-1,jy-1,jz-1) = u(0,0,0,jx,jy,jz);
});
TRANSLATING LOOPS TO KERNELS: KOKKOS

default target set at build time
asynchronous with host by default

Kokkos::parallel_for("internal corners",
Kokkos::MDRangePolicy<Kokkos::Rank<3>>({1,1,1},{mx,my,mz}),
KOKKOS_LAMBDA(const int jx, const int jy, const int jz) {
  u(0,0,0,jx,jy,jz) += u(nm1,0,0,jx-1,jy,jz)
  +u(0,nm1,0,jx,jy-1,jz)+u(nm1,nm1,0,jx-1,jy-1,jz)
  +u(0,0,nm1,jx,jy,jz-1)+u(nm1,0,nm1,jx-1,jy,jz-1)
  +u(0,nm1,nm1,jx,jy-1,jz-1)+u(nm1,nm1,nm1,jx-1,jy-1,jz-1);
  u(nm1,0,0,jx-1,jy,jz) = u(0,nm1,0,jx,jy-1,jz)
  = u(nm1,nm1,0,jx-1,jy-1,jz) = u(0,0,nm1,jx,jy,jz-1)
  = u(nm1,0,nm1,jx-1,jy,jz-1) = u(0,nm1,nm1,jx,jy-1,jz-1)
  = u(nm1,nm1,nm1,jx-1,jy-1,jz-1) = u(0,0,0,jx,jy,jz);
});
TRANSLATING LOOPS TO KERNELS: KOKKOS

**string name for Kokkos' built-in profiling**

```cpp
Kokkos::parallel_for("internal corners",
    Kokkos::MDRangePolicy<Kokkos::Rank<3>>({1,1,1},{mx,my,mz}),
    KOKKOS_LAMBDA(const int jx, const int jy, const int jz) {
        u(0,0,0,jx,jy,jz) += u(nm1,0,0,jx-1,jy,jz)
        +u(0,nm1,0,jx,jy-1,jz)+u(nm1,nm1,0,jx-1,jy-1,jz)
        +u(0,0,nm1,jx,jy,jz-1)+u(nm1,0,nm1,jx-1,jy,jz-1)
        +u(0,nm1,nm1,jx,jy-1,jz-1)+u(nm1,nm1,nm1,jx-1,jy-1,jz-1);
        u(nm1,0,0,jx-1,jy,jz) = u(0,nm1,0,jx,jy-1,jz)
        = u(nm1,nm1,0,jx-1,jy-1,jz) = u(0,0,nm1,jx,jy,jz-1)
        = u(nm1,0,nm1,jx-1,jy,jz-1) = u(0,nm1,nm1,jx,jy-1,jz-1)
        = u(nm1,nm1,nm1,jx-1,jy-1,jz-1) = u(0,0,0,jx,jy,jz);
    });
```
TRANSLATING LOOPS TO KERNELS: KOKKOS

triply nested loops
automatic mapping of loops to hardware

Kokkos::parallel_for("internal corners",
Kokkos::MDRangePolicy<Kokkos::Rank<3>>({1,1,1},{mx,my,mz}),
KOKKOS_LAMBDA(const int jx, const int jy, const int jz) {
    u(0,0,0,jx,jy,jz) += u(nm1,0,0,jx-1,jy,jz)
    + u(0,nm1,0,jx,jy-1,jz) + u(nm1,nm1,0,jx-1,jy-1,jz)
    + u(0,0,nm1,jx,jy,jz-1) + u(nm1,0,nm1,jx-1,jy,jz-1)
    + u(0,nm1,nm1,jx,jy-1,jz-1) + u(nm1,nm1,nm1,jx-1,jy-1,jz-1);
    u(nm1,0,0,jx-1,jy,jz) = u(0,nm1,0,jx,jy-1,jz)
    = u(nm1,nm1,0,jx-1,jy-1,jz) = u(0,0,nm1,jx,jy,jz-1)
    = u(nm1,0,nm1,jx-1,jy,jz-1) = u(0,nm1,nm1,jx,jy-1,jz-1)
    = u(nm1,nm1,nm1,jx-1,jy-1,jz-1) = u(0,0,0,jx,jy,jz);
});
TRANSLATING LOOPS TO KERNELES: KOKKOS

loop body as lambda

Kokkos::parallel_for("internal corners",
Kokkos::MDRangePolicy<Kokkos::Rank<3>>({1,1,1},{mx,my,mz}),
KOKKOS_LAMBDA(const int jx, const int jy, const int jz) {
  u(0,0,0,jx,jy,jz) += u(nm1,0,0,jx-1,jy,jz)
  +u(0,nm1,0,jx,jy-1,jz)+u(nm1,nm1,0,jx-1,jy-1,jz)
  +u(0,0,nm1,jx,jy,jz-1)+u(nm1,0,nm1,jx-1,jy,jz-1)
  +u(0,nm1,nm1,jx,jy-1,jz-1)+u(nm1,nm1,nm1,jx-1,jy-1,jz-1);
  u(nm1,0,0,jx-1,jy,jz) = u(0,nm1,0,jx,jy-1,jz)
    = u(nm1,nm1,0,jx-1,jy-1,jz) = u(0,0,nm1,jx,jy,jz-1)
    = u(nm1,0,nm1,jx-1,jy,jz-1) = u(0,nm1,nm1,jx,jy-1,jz-1)
    = u(nm1,nm1,nm1,jx-1,jy-1,jz-1) = u(0,0,0,jx,jy,jz);
});
TRANSLATING LOOPS TO KERNELS: KOKKOS

Kokkos::parallel_for("internal corners",
Kokkos::MDRangePolicy<Kokkos::Rank<3>>({1,1,1},{mx,my,mz}),
KOKKOS_LAMBDA(const int jx, const int jy, const int jz) {
    u(0,0,0,jx,jy,jz) += u(nm1,0,0,jx-1,jy,jz)
    +u(0,nm1,0,jx,jy-1,jz)+u(nm1,nm1,0,jx-1,jy-1,jz)
    +u(0,0,nm1,jx,jy,jz-1)+u(nm1,0,nm1,jx-1,jy,jz-1)
    +u(0,nm1,nm1,jx,jy-1,jz-1)+u(nm1,nm1,nm1,jx-1,jy-1,jz-1);
    u(nm1,0,0,jx-1,jy,jz) = u(0,nm1,0,jx,jy-1,jz)
    = u(nm1,nm1,0,jx-1,jy-1,jz) = u(0,0,nm1,jx,jy,jz-1)
    = u(nm1,0,nm1,jx-1,jy,jz-1) = u(0,nm1,nm1,jx,jy-1,jz-1)
    = u(nm1,nm1,nm1,jx-1,jy-1,jz-1) = u(0,0,0,jx,jy,jz);
});
TRANSLATING LOOPS TO KERNELS: RAJA

similar to Kokkos
body in lambda
RAJA::View operators

```cpp
template<typename T>
RAJA::kernel<for3async>(
    RAJA::make_tuple(r1mx, r1my, r1mz),
    [=] RAJA_DEVICE (const int jx, const int jy, const int jz) {
        u(jz, jy, jx, 0, 0) += u(jz, jy, jx-1, 0, 0, nm1)
        + u(jz, jy-1, jx, 0, nm1, 0) + u(jz, jy-1, jx-1, 0, nm1, nm1)
        + u(jz-1, jy, jx, nm1, 0, 0) + u(jz-1, jy, jx-1, nm1, 0, nm1)
        + u(jz-1, jy-1, jx, nm1, nm1, 0) + u(jz-1, jy-1, jx-1, nm1, nm1, nm1);
        u(jz, jy, jx-1, 0, 0, nm1) = u(jz, jy-1, jx, 0, nm1, 0)
        = u(jz-1, jy-1, jx-1, 0, nm1, nm1) = u(jz-1, jy, jx, nm1, 0, 0)
        = u(jz-1, jy, jx-1, nm1, 0, nm1) = u(jz-1, jy-1, jx, nm1, nm1, 0)
        = u(jz-1, jy-1, jx-1, nm1, nm1, nm1) = u(jz, jy, jx, 0, 0, 0);
    });
```
TRANS bulling loops to kernels: RAJA

execution policy and ranges defined in user code and reused

RAJA::kernel<for3async>(
    RAJA::make_tuple(r1mx, r1my, r1mz),
    [=] RAJA_DEVICE (const int jx, const int jy, const int jz) {
        u(jz, jy, jx, 0, 0, 0) += u(jz, jy, jx-1, 0, 0, nm1)
        + u(jz, jy-1, jx, 0, nm1, 0) + u(jz, jy-1, jx-1, 0, nm1, nm1)
        + u(jz-1, jy, jx, nm1, 0, 0) + u(jz-1, jy, jx-1, nm1, 0, nm1)
        + u(jz-1, jy-1, jx, nm1, nm1, 0) + u(jz-1, jy-1, jx-1, nm1, nm1, nm1);
        u(jz, jy, jx-1, 0, 0, nm1) = u(jz, jy, jx-1, 0, nm1, 0)
        = u(jz, jy-1, jx-1, 0, nm1, nm1) = u(jz-1, jy, jx-1, 0, nm1, 0, 0)
        = u(jz-1, jy, jx-1, nm1, 0, nm1) = u(jz-1, jy-1, jx, nm1, nm1, 0, 0)
        = u(jz-1, jy-1, jx-1, nm1, nm1, nm1) = u(jz, jy, jx, 0, 0, 0);
    });
TRANSLATING LOOPS TO KERNELS: RAJA

execution policy and ranges defined in user code and reused

// in header
using for3async = RAJA::KernelPolicy<GPU_KERNEL_ASYNC<
  RAJA::statement::For<2,GPU_BLOCK_Y_LOOP,
  RAJA::statement::For<1,GPU_BLOCK_X_LOOP,
  RAJA::statement::For<0,GPU_THREAD_X_LOOP,
  RAJA::statement::Lambda<0>>>>>>;

// in caller
const RAJA::RangeSegment r1mx(1,mx_);
const RAJA::RangeSegment r1my(1,my_);
const RAJA::RangeSegment r1mz(1,mz_);
TRANSLATING LOOPS TO KERNELS: RAJA

explicit mapping of loops to implementation

// in header
using for3async = RAJA::KernelPolicy<GPU_KERNEL_ASYNC<
    RAJA::statement::For<2,GPU_BLOCK_Y_LOOP,
    RAJA::statement::For<1,GPU_BLOCK_X_LOOP,
    RAJA::statement::For<0,GPU_THREAD_X_LOOP,
    RAJA::statement::Lambda<0>>>>>>;

// in caller
const RAJA::RangeSegment r1mx(1,mx_);
const RAJA::RangeSegment r1my(1,my_);
const RAJA::RangeSegment r1mz(1,mz_);
GPU portability through #ifdefs

// in header
using for3async = RAJA::KernelPolicy<GPU_KERNEL_ASYNC<
  RAJA::statement::For<2, GPU_BLOCK_Y_LOOP,
  RAJA::statement::For<1, GPU_BLOCK_X_LOOP,
  RAJA::statement::For<0, GPU_THREAD_X_LOOP,
  RAJA::statement::Lambda<0>>>>>>>>>>>;

// in caller
const RAJA::RangeSegment r1mx(1,mx_);
const RAJA::RangeSegment r1my(1,my_);
const RAJA::RangeSegment r1mz(1,mz_);
TRANSLATING LOOPS TO KERNELS: RAJA

**GPU portability through #ifdefs**

```c
#ifdef RAJA_ENABLE_HIP
#define GPU_BLOCK_X_LOOP RAJA::hip_block_x_loop
#define GPU_BLOCK_Y_LOOP RAJA::hip_block_y_loop
#define GPU_BLOCK_Z_LOOP RAJA::hip_block_z_loop
#define GPU_KERNEL_ASYNC RAJA::statement::HipKernelAsync
...
#else defined RAJA_ENABLE_CUDA
#define GPU_BLOCK_X_LOOP RAJA::cuda_block_x_loop
#define GPU_BLOCK_Y_LOOP RAJA::cuda_block_y_loop
#define GPU_BLOCK_Z_LOOP RAJA::cuda_block_z_loop
#define GPU_KERNEL_ASYNC RAJA::statement::CudaKernelAsync
...
#endif
```
FEATURE COMPARISON

- Multi-dimensional arrays
- Elements stored on GPUs
- MPI buffers stored on GPUs
- Translating loops to kernels
- **Running kernels in parallel**
RUNNING KERNELS IN PARALLEL

- Running multiple kernels at the same time on a single GPU (while running asynchronously with host)
- Cuda, Hip: launch kernels on different streams
- OpenMP

```c
!$omp target team depend(out:xface_) nowait
#pragma omp target teams depend(out:xfs0) nowait
```

- Don't try to identify all actual variable dependencies
- Think of `depend` target as virtual stream identifier
- Like a table translates variable addresses to GPU streams
- Raja: no explicit support yet
**RUNNING KERNELS IN PARALLEL: KOKKOS**

Construct execution spaces associated with different GPU streams (can be class member variables)

```cpp
Kokkos::DefaultExecutionSpace inner(innerStream);
Kokkos::DefaultExecutionSpace outer(outerStream);
...
Kokkos::parallel_for("faces",
    Kokkos::MDRangePolicy<Kokkos::Rank<3>>(outer,{0,0,0},{n,n,mm}),
    ...
    });
...
Kokkos::parallel_for("internal corners",
    Kokkos::MDRangePolicy<Kokkos::Rank<3>>(inner,{1,1,1},{mx,my,mz}),
    ...
    });
```
RUNNING KERNELS IN PARALLEL: KOKKOS

Kokkos::DefaultExecutionSpace inner(innerStream);
Kokkos::DefaultExecutionSpace outer(outerStream);
...
Kokkos::parallel_for("faces",
Kokkos::MDRangePolicy<Kokkos::Rank<3>>(outer,{0,0,0},{n,n,mm}),
    ...
});
...
Kokkos::parallel_for("internal corners",
Kokkos::MDRangePolicy<Kokkos::Rank<3>>(inner,{1,1,1},{mx,my,mz}),
    ...
});

*Kernels launched with different execution spaces can run concurrently on the same GPU*
FEATURE COMPARISON

- Multi-dimensional arrays
- Elements stored on GPUs
- MPI buffers stored on GPUs
- Translating loops to kernels
- Running kernels in parallel
OTHER TOPICS TO CONSIDER

- Mapping MPI tasks to GPUs
- Fusing kernel launches
- GPU synchronization for MPI sends
- Translating loops to Cuda and Hip kernels
- Shmem communication inside kernels
**EXTERNAL LINKS**

- OpenMP Specifications: [https://www.openmp.org/specifications/](https://www.openmp.org/specifications/)
- Cuda Toolkit Documentation: [https://docs.nvidia.com/cuda/](https://docs.nvidia.com/cuda/)
THANK YOU

Trey White
Frontier Center of Excellence
trey.white@hpe.com